A Performance Evaluation of Memory Hierarchy in Embedded Systems

Aleksandar Milenkovic, Milena Milenkovic, Nelson Barnes
Electrical and Computer Engineering Department
The University of Alabama in Huntsville
Email: {milenka | milenkm | nbarnes}@ece.uah.edu
URL: http://www.ece.uah.edu/~milenka

***Abstract*—The increasing speed gap between processors and memory makes the design of memory hierarchy one of the critical issues in general purpose embedded systems. As memory requirements for embedded applications grow, especially in emerging area of handheld multimedia devices, cache memories become crucial for providing high performance and reducing power. This paper describes a performance evaluation of typical cache design issues such as cache size and organization, block size, and replacement policy. The evaluation is done using simulation tools for architectural exploration based on ARM instruction set and MiBench benchmark suite. Our performance evaluation includes monitoring of dynamic cache behavior, since embedded systems designers are interested not only in the total number of cache misses, but also in the number of cache misses throughout application execution.**

I. INTRODUCTION

Continual advancement in semiconductor technology has provided remarkable gains in processor performance of approximately 1.6 times a year. This dramatic rise in performance poses requirements for a faster and larger memory system. Even though DRAM chips have become denser, their speed, improving only 7% a year, has not kept pace with the processor cycle time. In order to alleviate the processor-memory speed gap, computer designers rely on a cache hierarchy consisting of one or more levels of caches – each one smaller, faster, and more expensive per byte than the next level.

Embedded systems are the fastest growing portion of the computer industry and have recently become a focus of computer architecture research. In these systems, performance requirements are accompanied with strong requirements for reducing the cost of the system, which further transfers in the need to minimize the size of memory. Designers often face a real-time performance requirement where an application or its segment has an absolute time allowed for execution. Finally, for almost all battery-operated systems the requirement for reducing total energy consumed is critical.

Typical general-purpose mid- to high-end embedded processors are equipped with on-chip instruction and data caches interfacing larger and slower off-chip memories [1]. Cache hits usually take one or two processor clock cycles, while cache misses take tens of processor clock cycles and this speed gap will continue to grow. Here, caches are crucial not only in reducing memory latency, but also in reducing power hungry off-chip communication.

The goal of this paper is to offer a comprehensive performance evaluation of the main cache design issues in general purpose embedded processors, such as split versus unified cache, cache organization and size, block size, and replacement policy. As a performance metric, most studies use just the number of cache misses at the end of application execution or measurement period to evaluate cache performance. In this paper, we went one step further and measured the number of cache misses dynamically, per each 100K instructions. The goal of such an approach is to give us more insight into application behavior in the particular cache organization. The performance evaluation has been done using the ARM version of the SimpleScalar simulator toolset [2], executing recently developed MiBench benchmark suite [3], which represents a wide range of embedded applications.

The rest of the paper is organized as follows. Section II gives the problem description. Section III describes the experimental methodology including both simulation environment and benchmark programs used in the performance evaluation. Section IV provides the experimental results and Section V concludes.

II. PROBLEM DESCRIPTION

The primary parameters that determine the cache performance are cache and block size, cache write policy, block placement policy, and cache block replacement policy [4]. The write policy determines whether or not data should be forwarded to main memory on every write operation (write-through vs. write-back, respectively), and whether or not the cache blocks are allocated on write misses (write-allocate vs. write-no-allocate). Placement policies vary from direct-mapped to set-associative and fully associative. With direct-mapped caches, each memory block is mapped to a unique cache block, whether the cache block is empty or not. With fully associative cache memory, a memory block can be mapped to any of the empty cache blocks, if one exists. If there are no empty cache blocks, a replacement policy is used to select a cache block to be replaced. A set-associative cache memory divides the cache into sets and allows a memory block to be mapped to any of the empty cache blocks within the set the memory block is mapped to. If all cache blocks within the set are full, a replacement policy will select one of the cache blocks for replacement.

Numerous techniques have been proposed to further improve the efficiency of cache memories in both high-end and embedded processors [5]. Some well known compiler

0-7803-7697-8/03/$17.00 © 2003 IEEE.

optimization techniques are blocking, code placement methods, and software-directed block replacement polices. Blocking combines strip-mining and loop permutation, while code placement methods take advantage of the fact that embedded processors execute the limited set of programs. Hardware techniques such as multilevel caches, victim caches, write-buffers, way prediction, etc., can be employed to further improve cache efficiency.

As cache associativity in modern processors increases [1], it is important to revisit the effectiveness of common cache replacement policies. Ideally, when the cache controller must replace a cache block, it discards a cache block that will not be needed in the near future. However, the cache controller can only guess which cache block should be discarded. An optimal replacement (OPT) algorithm would replace a cache memory block whose next reference is the farthest away in the future among all cache memory blocks presently in the set [6]. This policy requires perfect knowledge of all future block references, and hence its implementation is infeasible. Instead, heuristics have to be used to determine which block is the most suitable to be replaced.

The state-of-the-art processors employ various policies such as Random, LRU (Least Recently Used), FIFO, and pLRU (pseudo LRU), indicating that there is no common wisdom about the best cache replacement policy. All these mechanisms, except Random, determine which cache memory block to replace by looking only at the cache memory past references. LRU replacement uses a heuristic that the cache block which has been accessed most recently will most likely be accessed again in the near future and accordingly, the cache block accessed "least recently" should be replaced by the cache controller. Although this technique is relatively efficient, it requires a number of status bits to track when each cache block is accessed – $Nways*log_2(Nways)$ bits are used for a set, where *Nways* is the number of cache blocks per set. To reduce the cost and complexity of LRU policy, Random policy can be used, but potentially at the expense of performance. Several researchers and computer designers have considered these two heuristics as too extreme in terms of implementation cost and performance. They have proposed various pLRU heuristics to reduce the hardware cost by approximating the LRU mechanism.

In the tree-based pseudo LRU (pLRUt) replacement heuristic, *Nway-1* bits are used to track the accesses to the cache blocks in a set. For example, in a 4-way set pLRUt track bits B0, B1, B2 are used to form a decision binary tree. The track bit B1 indicates whether two lower cache blocks CL0 and CL1 (B1=1), or 2 higher cache blocks CL2 and CL3 (B1=0) have been used recently. The track bit B0 determines further which one of two blocks CL0 (B0=1) or CL1 (B0=0) have been used recently; bit B2 tracks the access between cache lines CL2 and CL3. On a cache miss, bit B1 determines where to look for the least recently block, and B0 or B2 determines the least recently used block.

The other implementation of pLRU is based on using the most recently used (MRU) bits (pLRUm). In this case each cache block is assigned an MRU bit – *Nway* bits per set. The MRU bit for each cache block is set to a "1" each time a cache hit occurs on the cache block, indicating that the cache block has recently been used. When the cache controller is forced to replace a cache block, it examines the MRU bit for each cache block looking for a "0". When a "0" is found, it replaces that cache block and then sets it to a "1". A problem could occur if all MRU bits are set to a "1". If this happens, all blocks are unavailable for replacement causing a deadlock. To prevent this type of deadlock, all MRU bits in the set are cleared except the MRU bit being accessed when a potential overflow situation is detected.

III. EXPERIMENTAL SETUP

The performance evaluation of different cache organizations is done using *sim-cache* and *sim-cheetah* simulators from ARM version of the SimpleScalar toolset [2]. The original simulators have been modified to support additional pseudo-LRU replacement policies and collect corresponding statistics. In order to allow tracking of the dynamic behavior of caches, the *sim-cache* simulator has been modified to print interval statistics per specified number of instructions.

As a simulation workload, we use a subset of benchmarks taken from the freely available MiBench suite [3], which includes embedded applications and corresponding data sets. The MiBench applications are divided into six suites targeting the wide range of embedded applications, such as Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security, and Telecom. MiBench provides small and large data sets. In this study we use large data sets since they provide more realistic application workload. TABLE I gives a list of applications and a short description of each, as well as the total number of instructions executed (IC) and the number of load/store instructions executed (Refs).

TABLE I. MiBench Benchmarks.
(IC – the total number of instructions executed, Refs – the number of load/store instructions executed.)

Benchmark	**Type**	**Description**	**IC [$\times 10^6$]**	**Refs [$\times 10^6$]**
Adpcm	Telecom	ADPC Modulation	732.5	106.9
Blowfish	Security	Encription/decription	544.1	388.5
Cjpeg	Office	Image compression	104.6	39.8
DJpeg	Office	Image decompression	23.4	11.2
Dijkstra	Network	Shortest path problem	272.6	118.6
FFT	Telecom	Fast Fourier transform	301.8	130.2
Ghostscript	Office	Postcript interpreter	711	388.7
Lame	Consumer	MP3 encoder	1,151.8	631.6
Mad	Consumer	MPEG audio decoder	286.8	111.4
Patricia	Network	Routing	640.4	268.1
Qsort	Auto/Ind.	Sorting of strings	737.9	180.1
Rijndael	Security	Encription/decription	320.0	185.3
Sha	Security	Hashing	140.9	36.6
Susan	Auto/Ind.	Image recognition	29.8	9.8

We have considered two cache memory setups: with split first level instruction and data caches (L1I + L1D), and with unified first level cache (L1U) serving both instruction and data requests (Fig. 1). For each memory setup we have run simulations varying cache parameters such as:

- cache size - ranging from 0.5KB to 32KB;
- cache associativity – direct mapped (1w), 2-way (2w), 4-way (4w), and 8-way set-associative;
- cache replacement policies – FIFO, Random, LRU, pLRUt, pLRUm, and Optimal;
- cache block size – 32B, 64B.

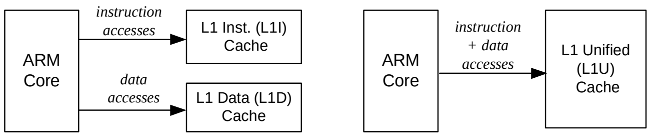

Fig. 1. Memory hierarchy.

As a measure of cache performance we use the number of misses per 1000 instructions. The relationship between this measure and more traditional cache miss rate is shown in (1).

$$Misses\ per\ 1000\ instructions = 1000 \times Miss\ rate \times \frac{Memory\ accesses}{Instruction\ count} \quad (1)$$

IV. EXPERIMENTAL RESULTS

The first set of experiments concentrates on two cache setups with split instruction and data caches and with a unified first level cache. Fig. 2, Fig. 3, and Fig. 4 show the number of cache misses per 1000 instructions for L1D, L1I, and L1U caches, respectively. The data have been collected on a direct-mapped cache memory with a block size of 32B. MiBench benchmarks show large variations in the frequency of memory operations (TABLE I). Some benchmarks like *GS*, *Lame*, *Rijndael* contain more than 50% of memory operations, *Admpc-encode* contain very few, while others contain about 40% memory operations. The number of cache misses varies significantly across applications, but most of them fit into caches of 16KB and 8KB, for both instruction and data.

TABLE II shows the average number of cache misses per 1000 instructions, when varying cache associativity (DM or 1w, 2w, 4w, 8w) and cache size (0.5KB – 32KB). For the direct-mapped split instruction and data caches, the combined number of misses is slightly lower compared to the number of misses in the corresponding unified cache of equivalent size – overall the difference is about 5% on average across all cache sizes in the favor of split caches. This difference is significant only for relatively small caches with 1KB combined size (0.5KB + 0.5KB). However, set-associative unified caches always have the lower number of cache misses – for 5% in the case of a 2-way, and about 8-9% for 4-way and 8-way caches. If other design constraints and requirements impose using direct cache memory organization (e.g., in order to achieve one clock cycle cache hit), the split organization is better than unified.

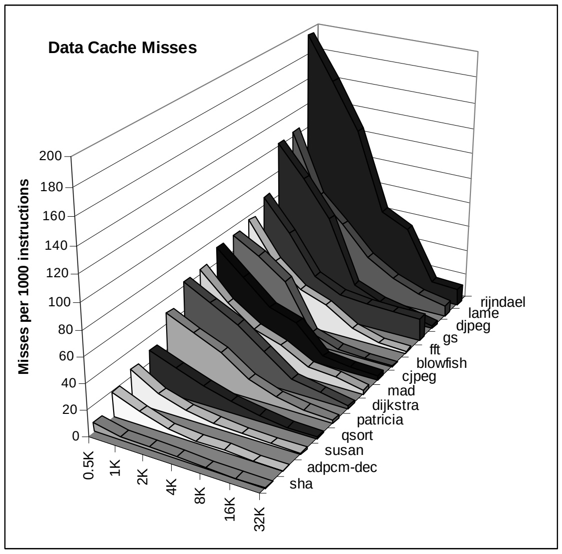

Fig. 2. Data cache misses per 1000 instructions.

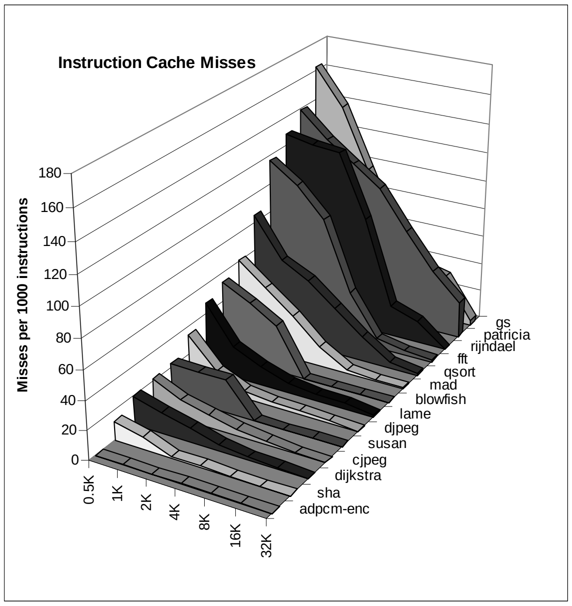

Fig. 3. Instruction cache misses per 1000 instructions.

The experiments also show that higher set-associativity is very beneficial in reducing the number of cache misses (see TABLE II). A 2-way set-associative cache, on average across all applications and cache sizes, reduces the number of misses for 27% for L1D, 27% for L1U, and 12% for L1I when compared to a direct-mapped cache. The benefit of a 4-way organization compared to direct-mapped is 36% for L1D, 14% for L1I, and 34% for L1U, while 8-way cache further reduces the number of cache misses, providing improvement of 40% for data, 14% for instruction, and 36% for unified caches. The results suggest that increasing the set-associativity is more useful in the case of data and unified caches than for instruction caches. The results of this study and the implementation complexity of highly associative caches give little justification for having 8-way caches and above since performance benefits are negligible.

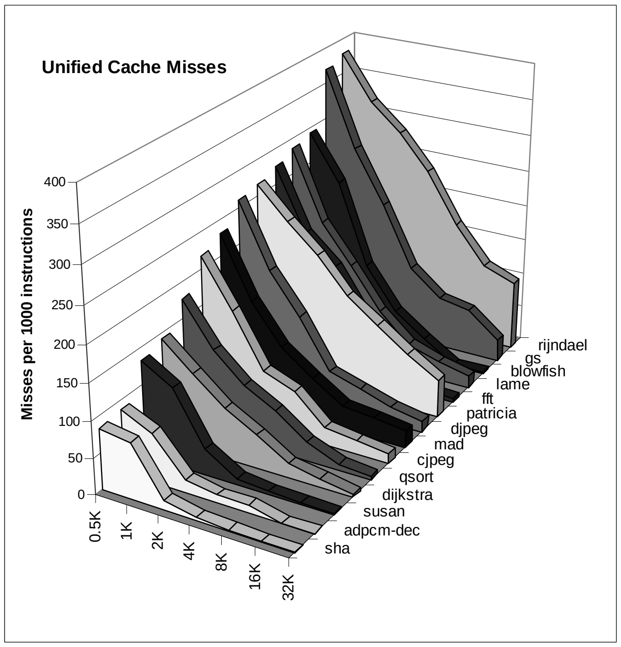

Fig. 4. Unified cache misses per 1000 instructions.

The absolute number of misses reduced by introducing set-associative organization is larger for smaller caches since conflict misses dominate, while the relative improvement is larger for larger caches.

Although our results suggest the use of unified cache for both data and instructions, in further evaluation we consider separate instruction and data caches, in order to better capture different instruction and data behavior for embedded applications.

The experiments aimed to explore effectiveness of the cache replacement policies suggest that pseudo LRU techniques are very good approximations of the true LRU. Furthermore, pLRUm outperforms LRU for most applications across all cache sizes and organizations. Random replacement policy performs almost consistently better than LRU and pseudo LRU techniques for instruction caches, while LRU replacement mostly works better than Random for data caches (TABLE III). FIFO replacement policy shows poor performance, although it is more costly than Random. The overall conclusion is that the use of Random policy is completely justifiable in instruction caches, while data caches need some more complex technique as pLRUm.

It is interesting to note that for some applications and some replacement techniques, the number of misses actually grows with increased associativity, due to more conflict misses and unoptimal replacement.

For both data and instruction cache, Optimal policy performs significantly better than the best of considered replacement policies, even in the two-way cache organization (TABLE II, TABLE III). Simple yet efficient replacement policies for embedded applications should be the matter of further investigation, especially for caches with the low degree of associativity.

For the two considered cache block sizes, 32B block size outperforms 64B for instruction caches, while 64B is better for data caches. In the unified cache organization, performance varies across applications. Fig. 5 shows misses for two block sizes (32B, 64B) for *Gs* benchmark.

Dynamic monitoring of cache behavior shows that some applications have execution regions with very low variations of the number of cache misses, e.g., *fft* and *gs*, while other applications have large but repeatable miss variations, e.g., *lame* and *djpeg*. Finally, some regions have completely irregular behavior, e.g., instructions in *cjpeg* (Fig. 6). These results could be used for dynamic re-sizing of caches, estimation of worst-case scenario, and power optimizations.

Due to space constraints we present only a subset of results. The complete performance evaluation results can be found on the following web site: http://www.ece.uah.edu/~milenka/arch/crp.htm.

TABLE II. Average cache misses per 1000 instructions when varying cache size and associativity. Legend: L1D – first level data cache, L1I – first level instruction cache, L1U – first level unified cache, SA – set associative, LRU – Least Recently Used replacement, OPT – Optimal replacement.

L1D	*DM*	*SA, LRU repl.*			*SA, OPT repl.*		
Size		*2w*	*4w*	*8w*	*2w*	*4w*	*8w*
0.5K	72.0	57.6	51.3	47.4	41.9	35.8	32.5
1K	53.0	39.7	34.6	32.0	28.8	23.9	21.1
2K	37.1	28.6	23.9	21.8	19.8	15.2	13.4
4K	19.6	14.3	14.0	13.2	9.3	7.6	6.3
8K	11.3	5.0	3.9	3.6	3.5	2.5	2.2
16K	5.8	2.3	1.7	1.6	1.7	1.2	1.1
32K	4.1	1.2	1.0	0.9	1.0	0.7	0.7
L1I	DM	SA, LRU repl.			SA, OPT repl.		
Size		2w	4w	8w	2w	4w	8w
0.5K	64.5	62.1	61.4	62.3	51.7	47.3	45.9
1K	51.9	47.9	48.4	49.1	38.3	34.5	33.2
2K	41.4	36.0	33.5	33.5	26.9	21.9	19.9
4K	23.5	21.3	21.3	20.5	13.9	10.6	9.2
8K	11.6	6.3	5.8	5.8	4.6	3.5	3.2
16K	7.6	3.2	3.1	3.3	2.3	1.6	1.3
32K	2.4	1.4	0.9	0.5	0.9	0.4	0.1
L1U	DM	SA, LRU repl.			SA, OPT repl.		
Size		2w	4w	8w	2w	4w	8w
0.5K	228.6	177.7	163.0	160.5	136.7	117.3	108.1
1K	168.8	123.0	110.8	107.5	97.4	81.5	75.2
2K	112.3	87.6	78.3	77.0	68.4	55.3	50.2
4K	72.0	56.6	52.5	49.9	41.9	33.2	28.8
8K	42.8	27.1	26.3	27.6	18.2	14.2	12.5
16K	27.8	11.4	7.8	7.5	7.6	4.6	4.0
32K	19.0	5.1	2.9	2.3	3.6	1.6	1.2

V. CONCLUSIONS

Cache memories have become widespread in general-purpose embedded processors. This paper presents the experimental results of the performance evaluation aimed to explore key cache design issues such as cache size and organization, block size, and replacement policy in general-purpose embedded processors.

The results suggest that for relatively small direct mapped caches, split instruction and data caches show better performance than equivalent unified cache. For set-associative caches, the unified cache almost always outperforms the split caches. Increasing cache associativity reduces the number of cache misses but the results show little justification for 8-way associative caches and above. Higher associativity brings more benefit to data and unified caches than to instruction caches. Pseudo-LRU techniques perform as well as true LRU. However, for instruction caches Random policy performs the best, while for data caches pLRUm shows the best performance. Relatively large difference between Optimal policy and the best unoptimal replacement policy suggest a need for further investigation in an attempt to close the gap.

TABLE III. Average cache misses per 1000 instructions varying replacement policy.

	Instr. cache			Data cache		
1K	2w	4w	8w	2w	4w	8w
fifo	58.2	58.6	59.2	47.9	43.7	41.0
random	56.8	56.4	56.7	48.3	44.0	41.0
lru	57.9	58.1	58.8	45.6	39.8	36.8
pLRUt	57.9	57.6	58.3	45.6	40.1	37.2
pLRUm	57.9	56.2	56.3	45.6	39.3	35.8
opt	46.7	42.1	40.5	32.6	27.2	24.1
2K	2w	4w	8w	2w	4w	8w
fifo	45.9	42.8	43.1	34.6	29.4	27.1
random	43.1	39.2	37.8	34.5	29.2	27.4
lru	45.5	42.1	42.2	33.3	27.8	25.4
pLRUt	45.5	41.7	40.8	33.3	27.9	26.0
pLRUm	45.5	40.2	38.5	33.3	27.3	25.1
opt	34.1	27.6	25.1	22.6	17.5	15.4
4K	2w	4w	8w	2w	4w	8w
fifo	27.6	27.6	26.2	17.5	17.1	16.2
random	23.5	20.6	18.9	16.9	16.0	14.8
lru	27.1	27.3	25.9	16.7	16.4	15.5
pLRUt	27.1	27.0	25.3	16.7	16.3	15.0
pLRUm	27.1	22.5	22.3	16.7	15.5	14.4
opt	17.7	13.7	11.9	10.5	8.7	7.2

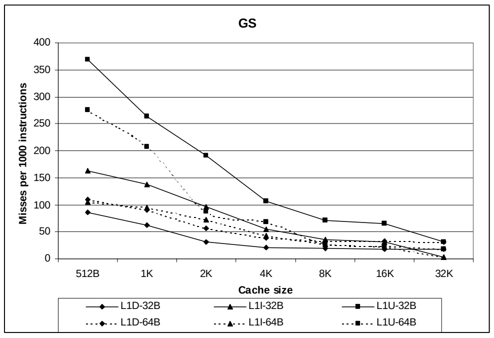

Fig. 5. Average misses per 1000 instruction when varying cache block size.

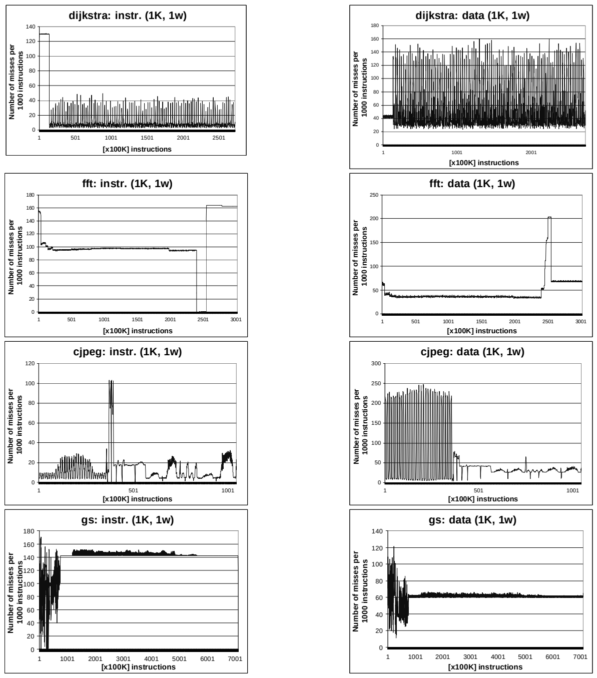

Fig. 6. Dynamic cache behavior for 1K, 32B, direct-mapped cache.

VI. REFERENCES

[1] Intel ® Xscale ™ Core – Developer's Manual, December 2000, http://developer.intel.com

[2] D. Burger, T. Austin, "The SimpleScalar Tool Set, Version 2.0," *University of Wisconsin-Madison Technical Report #1342*, June 1997.

[3] M. Guthaus, J. Ringenberg, T. Austin, T. Mudge, R. Brown, "MiBench: A free, commercially representative embedded benchmark suite, *in Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*," Austin, TX, December 2001. (http://www.eecs.umich.edu/mibench/).

[4] J. L. Hennessy, D. Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufmann Publishers, 2003.

[5] N. D. Dutt "Memory Organization and Exploration for Embedded Systems-on-Silicon," *Proc. 1997 International Conference on VLSI and CAD (ICVC'97)*, Oct. 1997.

[6] L.A. Belady, "A study of replacement algorithms for a virtual storage computer," *IBM Systems Journal*, 5(2):79-101, 1966.

[7] J. F Cantin and M. D. Hill, Cache Performance of the SPEC CPU2000 Benchmarks, http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/

[8] A. Malamy, R. Patel, N. Hayes, "Methods and aparatus for implementing a pseudo-LRU cache memory replacement scheme with a locking feature," *United States Patent 5353425*, October 1994.

[9] K. So, R. N. Rechtshaffen, "Cache operations by MRU change," *IEEE Transaction on Computers*, vol. 37, no. 6, pp. 700-707, June 1988.

[10] Z. Wang, K. McKinley, A. Rosenberg, C. Weems, "Using the Compiler to Improve Cache Replacement Decisions, " in *Proceeding of the International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, Virginia, September, 2002.